An Adaptable Binary Entropy Coder[1]

Aaron Kiely and Matthew Klimesh
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Mail Stop 238-420, Pasadena, CA 91109
e-mail: {aaron, klimesh}@shannon.jpl.nasa.gov

Abstract— We present a novel entropy coding technique which is based on recursive interleaving of variable-to-variable length binary source codes. The encoding is adaptable in that each bit to be encoded may have an associated probability estimate which depends on previously encoded bits. The technique can achieve arbitrarily small redundancy, and may have advantages over arithmetic coding, including most notably the admission of a simple and fast decoder. We discuss code design and performance estimation methods, as well as practical encoding and decoding algorithms.

1 Introduction

In data compression algorithms the need frequently arises to compress a binary sequence in which each bit has some estimated distribution, i.e., probability of being equal to zero. If long runs of bits have nearly identical distributions, then simple source codes, most notably Golomb's runlength codes [1, 2], are quite efficient. However, in many practical situations, not only does the distribution vary from bit to bit, but it is desirable to have the estimated distribution for a bit depend on the values of earlier bits. Accommodating such a dynamically changing probability estimate is tricky because the decoder must make the same estimates as the encoder. Thus, before the ith bit can be decoded, the value of the first $i-1$ bits must be determined. This complication makes it difficult to efficiently use simple source codes such as runlength codes.

To our knowledge, currently the only efficient encoding methods in this case are arithmetic coding [3, 4] and the relatively unknown technique called interleaved entropy coding [5], which is a generalization of the "block Melcode" [6]. In this paper, we describe a new entropy coding technique which is a generalization of the interleaved entropy coding method. The technique efficiently encodes a binary source with a bit-wise adaptive probability estimate by recursively encoding groups of bits with similar distributions, ordering the output in a way that is suited to the decoder. As a result, the decoder has low complexity.

As an indication of the interest in low complexity encoding and decoding of sequences with adaptive probability estimates, we note that much effort has been put into reducing the complexity of arithmetic coding [7, 8, 9, 10].

The functionality of our coding technique is essentially the same as that of binary arithmetic coding; however, our technique does not yield an arithmetic code and

[1]The work described was funded by the TMOD Technology Program and performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

there are many practical differences. Arithmetic encoding of one bit requires a few arithmetic operations and, unless approximations are made, at least one multiplication. Our encoder requires no arithmetic operations except those needed to choose a code index based on the bit distribution; however it requires some bookkeeping and bit manipulation operations. Our encoder requires more memory than arithmetic coding. Arithmetic decoders are generally of similar complexity to the encoders, but our decoder is much simpler than our encoder: it needs fewer operations than the encoder, and requires only a small amount of memory.

1.1 The Source Coding Problem

We examine the problem of compressing a sequence of bits $b_1, b_2, \ldots$ from a random source. The estimate of source probability $p_i = \text{Prob}[b_i = 0]$ may depend on the values of the source sequence prior to index i, and on any other information available to both the encoder and decoder. This dependence encompasses both adaptive probability estimation as well as correlations or memory in the source. Consequently, efficient encoding requires a bit-wise adaptable encoder. We are not concerned with methods of modeling the source, and so we make no distinction here between the actual and estimated source distributions.

Without loss of generality, we will assume that $p_i \geq 1/2$ for each index i. If this were not the case for some p_i, we could simply invert bit b_i before encoding to make it so.

We also assume that the decoder can determine when decoding is complete. In practice, this often occurs automatically, or straightforward methods can be used, such as transmitting the sequence length prior to the compressed sequence. Such methods will not be addressed here.

Although we only discuss the compression of binary sequences, given any nonbinary source we can assign prefix-free binary codewords to source symbols to produce a binary stream. Thus the technique can be applied to nonbinary sources as well.

1.2 The Recursive Interleaved Entropy Coding Concept

In this section we give an overview of how the entropy coding technique works and why it can give good performance. To simplify the explanation, some of the processing details are omitted until Section 2.

Since, by assumption, each bit has probability of zero at least 1/2, we are concerned with the probability region $[1/2, 1]$. We partition this region into several narrow intervals, and with each interval we associate a bin that will be used to store bits. When bit b_i arrives, we place it into the bin corresponding to the interval containing p_i. Because each interval spans a small probability range, all of the bits in a given bin have nearly the same probability of being zero, and we can think of each bin as corresponding to some nominal probability value.

For each bin (except the leftmost bin, which contains probability 1/2) we specify an exhaustive prefix-free set of binary codewords. When the bits collected in a bin form one of these codewords, we delete these bits from the bin and encode the value of the codeword by placing one or more new bits in other bins[2]. This process is conveniently described using a binary tree. Each codeword is assigned to a terminal node in the tree, internal nodes[3] are labeled with a destination bin, and the branch labels (each a zero or one) correspond to the output bits that are placed in the destination bins.

For example, Figure 1 shows a tree that might be used for a bin with nominal probability 0.9. The prefix-free codeword set for this bin is $\{00, 01, 1\}$, shown as labels of the terminal nodes in the tree. If the codeword to be processed in the bin is 00, which occurs with probability approximately 0.81, we place a zero in the bin that contains probability 0.81. If the codeword is 1, first we place a one in the bin containing probability 0.81, which indicates that the codeword is something other than 00, then we place a zero in the bin containing probability 0.53 because, given that the codeword is not 00, the conditional probability that the codeword is 1 is approximately 0.53. We can see that this process might contribute to data compression because the most likely codeword is 00, which is represented using a single bit.

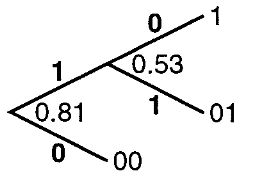

Figure 1: Example of a tree for a bin with representative probability 0.9.

For the leftmost bin we do not define a tree such as the one in Figure 1. Instead, bits in this bin form the encoder's output. Bits that reach the first bin have probability of being zero very close to 1/2 and are thus nearly incompressible, so transmitting these bits uncoded does not add much redundancy.

During the encoding process, bits arrive in various bins either directly from the source or as a result of processing codewords in other bins. Our goal is to have bits migrate to the leftmost (uncoded) bin, where they are transmitted. To accomplish this, we impose the constraint on the design of our trees that all new bits resulting from the processing of each codeword must be placed in bins strictly to the left of the bin in which the codeword was formed. Apart from our desire to move bits to the left, this constraint also prevents encoded information from traveling in "loops", which would make coding difficult or impossible. Thus if a bin has nominal probability p, we would like the probability of a zero for each output bit to be in the range $[1/2, p)$. A tree with this property is said to be *useful* at p. Perhaps surprisingly, useful trees exist everywhere:

Theorem *For any given probability value $p \in (1/2, 1)$, there exists a useful tree, i.e., one with the property that all output bits have probability of zero in the range $[1/2, p)$.*

We prove this by constructing an infinite family of trees for which at least one tree is useful at any $p \in (1/2, 1)$. Figure 2 illustrates this construction. We omit the details of the proof.

When we reach the end of the bit sequence to be encoded and no codewords remain in any bin, there will generally be partially formed codewords in one or more bins. Since these bits are needed for decoding, we append one or more extra bits to

[2]The ordering of the new bits in a bin is not straightforward and we save these details for Section 2.2.

[3]Throughout this paper we refer to any non-terminal node as an internal node.

each of these partial codewords to form complete codewords which are then processed in the normal manner[4].

We can see that some redundancy is present in this system because the bins have positive width — the probability associated with a bit that arrives in a bin will usually not exactly equal the bin's nominal probability, and bits in the leftmost bin are transmitted uncompressed even though they may not have probability of zero exactly equal to 1/2. As one might expect, however, by increasing the number of bins and/or the size of the trees, we can trade complexity for performance and decrease the maximum redundancy to arbitrarily small values.

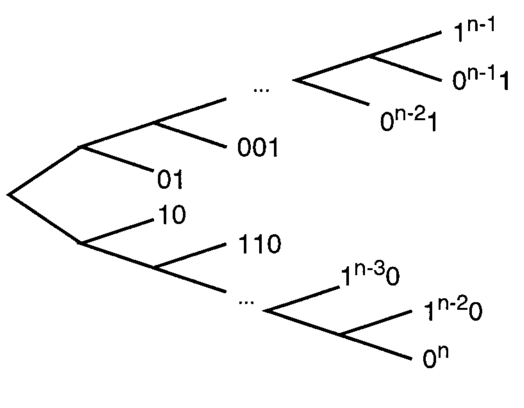

Figure 2: A tree that is useful for $p \in (\gamma_n, \gamma_{n-2})$, where $n \geq 2$. Here γ_i is the root in $(1/2, 1]$ of $p^i = (1-p)^{i-1}$ for $i > 1$, and $\gamma_0 = 1$.

In practice, the encoder and decoder do not keep track of probability values. Instead, each bin is assigned an index, starting with index 1 corresponding to the leftmost (uncoded) bin. At each internal node in the tree we identify the index, rather than the nominal probability value, of the bin to which the associated output bit is mapped. The constraint we impose on encoder design is that each output bit from the tree for bin j must be mapped to a bin with index strictly less than j. No computations involving probability values are needed apart from those which may be required to map each input bit b_i to the appropriate bin index.

For example, a five bin encoder is defined by the trees shown in Figure 3. Figure 3(c) indicates, e.g., that if codeword 01 is formed in bin 4 then we place bits 1,0,1 in bins 3,2,1 respectively. A complete encoder description also requires identification of the probability region over which each bin should be used (or a rule for mapping input bits to bins), but we omit this detail to simplify the discussion.

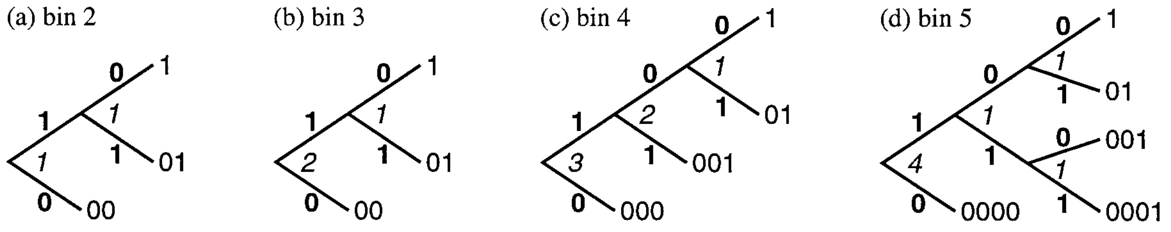

Figure 3: A possible design for a five bin encoder. The first bin is uncoded, hence no tree is shown. Bin indices are shown in italics, output bits in boldface. The input bits are the codewords shown as terminal nodes of each tree.

[4]The method of selecting these extra bits that "flush" the encoder is relatively unimportant, and we omit the details in this paper.

1.3 Relation to Interleaved Entropy Coding

An important special case of the entropy coder arises when all output bits generated from each tree in the encoder are mapped to the uncoded bin. In this case the encoder amounts to interleaving several separate variable-to-variable length binary codes. Imposing this restriction on the encoder still allows for very low redundancy, and significantly reduces encoding complexity, so very fast encoding is possible. This technique was first suggested in [6], which used interleaved Golomb codes for compression. A more thorough analysis of interleaved entropy coding appears in [5].

2 Encoding and Decoding

2.1 Decoder Operation

We first describe decoding since it determines the encoding procedure. It is convenient to think of each bin in the decoder as containing a list of bits. To decode, we initially place all of the encoded bits in the first (uncoded) bin, and all other bins are empty. At any time, each nonempty bin (with the exception of the uncoded bin) will contain a single codeword or a prefix of a codeword. Decoding the next source bit amounts to taking the next bit from the bin to which the source bit was assigned. If this bin is empty, we first reconstruct the codeword in that bin by taking bits from other bins as needed.

Software decoding uses two recursive procedures, `GetBit` and `GetCodeword`. `GetBit` simply takes the next available bit from the indicated bin. If the bin is empty then it first calls `GetCodeword`. Given an empty bin, `GetCodeword` determines which codeword must have occupied the bin by taking bits from other bins (via `GetBit`), then places that codeword in the bin. The `GetCodeword` procedure is similar to Huffman decoding, except that at each step we take the next bit from the appropriate bin, not (necessarily) from the encoded bit stream.

To decode the ith bit, let `binindex` equal the index of the bin to which the ith bit would have been assigned. This assignment may be a function of all of the previously decoded bits and any other information available to both the encoder and decoder. Then the ith decoded bit is equal to `GetBit(binindex)`.

2.2 Encoder Operation

To ensure that decoding is possible, we must pay careful attention to the order in which bits are processed by the encoder. Processing bits in the correct order is not straightforward.

One encoding method that produces encoded bits in the appropriate order involves maintaining a linked list of bit values sorted in order of priority. Each record in the list stores the bit value and the index of the bin that contains the bit. Initially the list contains the entire input sequence in order of arrival. To encode, at each step we identify the nonempty bin with the highest index. We take bits (in priority order) from this bin until we have formed a codeword, appending flush bits if needed to complete the final codeword of the bin. We delete the bits that formed the codeword and insert the resulting output bits in the list at the location of the highest priority bit in the codeword.

For example, suppose the linked list for the encoder of Figure 3 is as shown in the left half of Figure 4. Bin 4 is the highest indexed nonempty bin, so we search through the list for bits in bin 4 until we form the codeword 01. This codeword produces output bits 1,0,1, in bins 3,2,1 respectively (see Figure 3(c)), so these records are inserted in the linked list as shown in the right half of Figure 4.

When all bits are in the first bin, the encoder output consists of these bits taken in priority order.

To manage long input sequences with limited memory, we can partition the input sequence into blocks of known size and encode each block separately. An alternative technique is not described here.

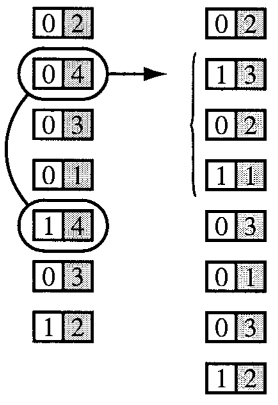

Figure 4: One step of encoding in software using the encoder of Figure 3. The shaded boxes indicate bin indices, unshaded boxes show bit values.

3 Estimating Rate

We would like to quantify the performance of a given encoder design. One metric we can estimate is the rate (the expected number of output bits per input bit) when the input to the encoder is an IID stream of bits into bin j, each bit having probability of zero equal to p. We denote this quantity by $R_j(p)$.

Since bins produce output bits that are placed in other bins, estimates of $R_j(p)$ generally rely on rate estimates for other bins. If bin ℓ has as input λ_1 bits with probability q_1 and λ_2 bits with probability q_2, the resulting contribution to rate might be approximated as

1. $\lambda_1 R_\ell(q_1) + \lambda_2 R_\ell(q_2)$, or
2. $(\lambda_1 + \lambda_2) R_\ell \left(\dfrac{\lambda_1 q_1 + \lambda_2 q_2}{\lambda_1 + \lambda_2} \right)$.

The first approximation would tend to be more accurate when long runs of bits in bin ℓ have the same probability, and the second would be more accurate if the two types of bits are well mixed. The first tends to be optimistic when the rate function is convex $\cap$, the latter tends to be optimistic when the rate function is convex $\cup$.

In this section we describe two recursive techniques for estimating $R_j(p)$ based on the above approximations. Both techniques usually give quite good estimates. The rate estimates produced are asymptotic as the input sequence length becomes large, i.e., the cost of bits used to flush the encoder is not included.

The rate estimation techniques do not give exact results because the rate functions are in general nonlinear, and because bits arriving in each bin may not be independent. This dependence arises because encoding a single codeword may result in several output bits being placed in the same bin.

3.1 First Method for Rate Estimation

We can estimate $R_j(p)$ recursively using the estimates for $R_1(p)$, $R_2(p), \ldots, R_{j-1}(p)$. If for each input bit internal node k of the tree for bin j produces $\eta_k(p)$ expected bits, each with probability of zero $q_k(p)$, then we use the estimate

$$R_j(p) = \begin{cases} \sum_k \eta_k(p) R_{b(k)}(q_k(p)), & j > 1 \\ 1, & j = 1 \end{cases} \tag{1}$$

where $b(k)$ is the output bin index for the kth node in the tree, and the sum is over all internal nodes in the tree.

For example, using (1) to estimate $R_4(p)$ for the encoder of Figure 3 gives

$$R_4(p) = (1-p)\left(4 - \frac{1+p}{2+2p+p^2} + \frac{1}{1-p^{12}}\right).$$

3.2 Second Method for Rate Estimation

In the second technique for estimating $R_j(p)$, for each bin we produce a list of (λ, q) pairs. Each pair in the list represents an expected number of bits λ and corresponding probability of zero q arising from the output of some higher indexed bin or from the source. Initially each list is empty except the list for bin j, which contains the pair $(1, p)$.

At each step, if the list for bin ℓ (initially $\ell = j$) contains pairs (λ_1, q_1), $(\lambda_2, q_2), \ldots$, (λ_m, q_m), we compute the total expected number of bits in the bin

$$\Lambda_\ell = \sum_k \lambda_k$$

and the average probability of a zero in the bin

$$Q_\ell = \frac{1}{\Lambda_\ell} \sum_k q_k \lambda_k.$$

Treating the input to bin ℓ as Λ_ℓ bits, each with probability of zero Q_ℓ, we compute the resulting pairs (λ'_k, q'_k) at each internal node in the tree and append (λ'_k, q'_k) to the list for the bin to which the output bit associated with node k is mapped.

We repeat this procedure, continuing to the first bin. Finally, our estimate of $R_j(p)$ is equal to the total expected number of bits in the first bin, Λ_1.

Using this technique to estimate $R_4(p)$ for the encoder of Figure 3 gives

$$R_4(p) = (1-p)\left(3 + \frac{(2-p^6)^2}{(3-p^2+p^3-p^5)(1-p^6)}\right).$$

For $p \in (1/2, 1)$, the method of (1) gives a slightly higher rate estimate in this example.

Variations of these techniques can be used to estimate the rate associated with a source that produces bits with varying (but known) distributions.

4 Code Design

In this section we illustrate a procedure to design an encoder. We begin with a redundancy target Δ which is the maximum allowed redundancy (in bits) and a set of candidate trees to be used in the encoder. In this context each tree does not include assignments of bin indices to internal nodes or output bit labels to branches. These assignments will be made as part of the design procedure. With a suitable set of candidate trees, we can produce an encoder that has redundancy less than Δ for arbitrarily small Δ.

In addition to the family of trees illustrated in Figure 2, there are many other useful trees we can use to design good codes. For example, Figure 5 shows the five useful trees with four terminal nodes.

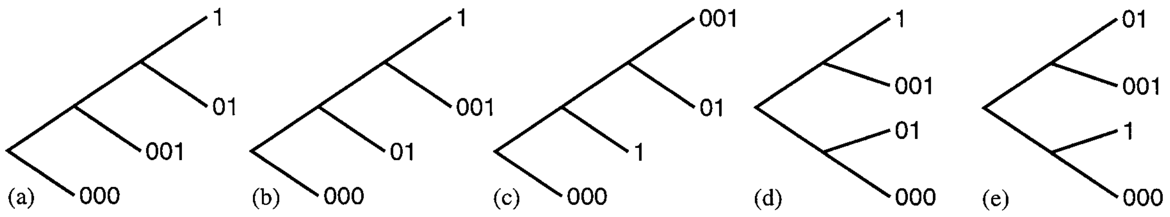

Figure 5: The set of useful trees with four terminal nodes. Tree (a) is useful for p in the range (0.755, 1), trees (b), (c), (d) are useful over (0.682, 1), and tree (e) is useful over (0.618, 1).

We select the trees for the bins in order of increasing bin index. When designs for bins $1, 2, \ldots, j-1$ have been completed, designing the jth bin amounts to selecting a tree for the bin, assigning bin indices to internal nodes and output bit labels to branches, and calculating z_{j-1}, the probability value where we switch from bin $j-1$ to bin j. (Of course no design work is required for the first bin since it is uncoded and $z_0 = 1/2$.) For example, Figure 6 shows a case where the encoder has been designed for the first three bins, and our redundancy target Δ is met when p is less than some value p^*. Thus we know that $z_3 \leq p^*$, and we need to specify the tree to use for the fourth bin of the encoder.

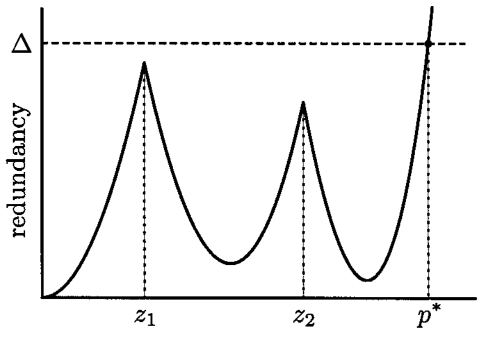

Figure 6: Redundancy of an encoder after designing the first three bins.

To do this, we can take from our set of candidate trees any tree that is useful at p^* and assign branch and internal node labels based on this probability value. That is, we calculate the branch probability for each internal node in the tree and label the branches so that a zero output is more likely than a one at each node. Then, at each internal node, if a zero output bit occurs with probability q, we map this bit to the bin with index ℓ such that $q \in [z_{\ell-1}, z_\ell)$.

This construction maps output bits to bins in regions where the redundancy is less than the target Δ, and it can be shown that (to the extent that (1) is accurate) the redundancy at probability p^* is strictly less than Δ. This follows in part from the following lemma (proof omitted):

Lemma *If a tree is useful at p^*, then the expected number of output bits is less than the expected number of input bits, or equivalently,*

$$\sum_k \lambda_k(p^*) < 1$$

(where $\lambda_k(p^*)$ is the expected number of output bits generated at node k and the sum is over all internal nodes k).

Thus, the tree we have selected for the new bin produces redundancy less than Δ at probability p^*. Since the rate functions for each bin are continuous, we have extended the range where the encoder meets the redundancy target.

We can also try assigning branch labels, and even selecting a tree, based on some probability target value larger than p^*. This alternative generally produces larger redundancy at p^*, but frequently meets the redundancy target Δ at p^* and may extend further the range over which bin j is used, which can help to reduce the total number of bins used in the encoder.

We have found some good encoders using the design technique described in this section. Figure 7 shows the redundancy of some of these designs, computed using the rate estimation technique of Section 3.2.

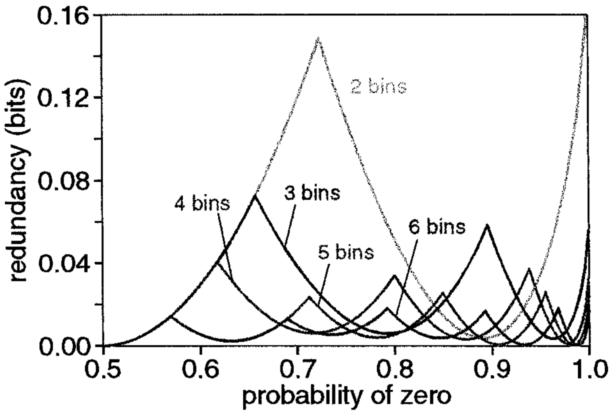

Figure 7: Redundancy of some encoders that use a small number of bins.

5 Conclusion

The techniques described here have been used to produce working software encoders and decoders that confirm the performance estimates shown. Low redundancy is attainable using relatively small trees (e.g., an average of 6 terminal nodes in the trees used for the encoders of Figure 7). This technique may be a viable alternative to arithmetic coding when decoding speed is important.

References

[1] S. W. Golomb, "Run-Length Encodings," *IEEE Transactions on Information Theory*, vol. IT-12, no. 3, pp. 399–401, July, 1966.

[2] R. G. Gallager and D. C. Van Voorhis, "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Transactions on Information Theory*, vol. IT-21, no. 2, pp. 228–230, March, 1975.

[3] J. Rissanen and G. G. Langdon, "Arithmetic Coding," *IBM Journal of Research and Development*, vol. 23, no. 2, pp. 149–162, March, 1979.

[4] I. H. Witten, R. M. Neal, J. G. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, June, 1987.

[5] P. G. Howard, "Interleaving Entropy Codes," *Proc. Compression and Complexity of Sequences 1997* pp. 45–55, 1998.

[6] F. Ono, S. Kino, M. Yoshida, and T. Kimura, "Bi-Level Image Coding with MELCODE — Comparison of Block Type Code and Arithmetic Type Code," *Proc. IEEE Global Telecommunications Conference (GLOBECOM '89)*, pp. 0255 - 0260, Nov. 1989.

[7] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic Coding Revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, July, 1998.

[8] L. Huynh and A. Moffat, "A Probability-Ratio Approach to Approximate Binary Arithmetic Coding," *IEEE Transactions on Information Theory*, vol. 43, no. 5, pp. 1658–1662, September, 1997.

[9] D. Chevion, E. D. Karnin, E. Walach, "High Efficiency, Multiplication Free Approximation of Arithmetic Coding," *Proc. IEEE Data Compression Conference*, pp. 43–52, April, 1991.

[10] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon, and R. B. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM Journal of Research and Development*, vol. 32, no. 6, pp. 717–726, November, 1988.